Vol.2022-SE-210 No.5
2022/3/11

JavaScriptにおける永続時変値の実現法の提案

日高 大地[1,a)]　紙名 哲生[1,b)]

概要：リアクティブプログラミング（RP）における時変値では，変数への明示的な再代入を行うかわりに，変数の値が時間とともに変化する．さらにJavaに基づくRP言語SignalJにおいては，時変値の値変化の履歴を時系列データとみなし，その値を指定された時刻の関数とする永続時変値の機構が提供される．永続時変値は現在SignalJでのみ実現されている機構であるが，その実体は言語処理系の外部にある時系列データベースであり，特定のプログラミング言語には依存しない設計となっている．本研究ではこの特徴を生かした，JavaScriptにおける永続時変値の実現方法について議論する．本提案ではSignalJと異なり，JavaScriptの言語拡張は行わず，JavaScriptのライブラリとして永続時変値の機構を実現する．

キーワード：リアクティブプログラミング，時系列データベース，WebSocket，Node.js

A method for realizing persistent time-varying values in JavaScript

Daichi Hidaka[1,a)]　Tetsuo Kamina[1,b)]

Abstract: A signal in reactive programming (RP) is considered as a function of time and its value is updated automatically without performing explicit assignments. Furthermore, the Java-based RP language SignalJ provides persistent signals, where the update history of a signal is considered time-series data, and the value of the signal is considered as a function of a specified (possibly past) time. Currently, this mechanism is only supported by SignalJ; however, the entity of a persistent signal exists in the time-series database, which exists in the outside of the language, and thus, this mechanism can be reused by the other language. In this paper, we propose a method for realizing persistent signals in JavaScript by utilizing this characteristic. Unlike SignalJ, in this implementation, persistent signals are realized not as a language extension but as a library executable on a standard JavaScript runtime.

Keywords: Reactive programming, Time-series databases, WebSocket, Node.js

1. はじめに

現在，リアクティブプログラミング（RP）と呼ばれる，時間と共に変化していくデータ（ストリーム）同士の依存関係や計算を宣言的に記述するプログラミングの手法が注目を集めている．RP言語の中には，時変値と呼ばれる時間変化する値を抽象化する変数を使用できるものが存在する．実際に時変値を使ったセンサとモータの連携を考える．モータの操作量はセンサの値で決まるとする．このとき，センサの値とモータの操作量（いずれも変数）を時変値として表現すると，センサの値が変わると自動的にモータの操作量も変化する．

一方，IoTのアプリケーションなどの場面では，時間変化する値を過去に遡って求めたい場合がある．例えば車輌の走行履歴を追跡するプログラムを考える．ある車輌が事故を起こしたり，危険な走行を行なった場合には，それらが発生した時点に遡って値を求め，何らかの検証を行う．時変値においてそのような演算を考えるとき，時変値の更新履歴が必要になる．

そこで，時変値の値変化の履歴を時系列データとみなし，その値を指定された時刻の関数とする永続時変値 [5], [6] の

[1] 大分大学理工学部共創理工学科知能情報システムコース
Division of Computer Science and Intelligent Systems, Oita University, Dannoharu 700, Oita 870-1192, Japan
[a)] v1858251@oita-u.ac.jp
[b)] kamina@acm.org

機構が Java に基づく RP 言語 SignalJ[4] において提案された．永続時変値では，ある時刻における時変値とそれが含まれるデータフローの中のそれぞれの値は，単にその時刻を指定することで求まる．この機構は研究が始まったばかりであり，現在では SignalJ 言語でのみ実装されている．

一方で，Web アプリケーションにおけるフロントエンドは，Java ではなく JavaScript が用いられており，そのようなアプリケーションで永続時変値を用いたいという需要は考えられる．幸い，永続時変値の実体は言語処理系の外部にある時系列データベースであり，特定のプログラミング言語には依存しない設計となっているため，その仕組みを再利用できる．

本研究では永続時変値の機構を JavaScript でも実現する方法を提案する．永続時変値の実体は時系列データベースにあるので，時系列データベースへの問い合わせを行う機構を JavaScript で実装できるのであれば永続時変値の機構を実装することができる．しかしながら，そこには以下に示す解決すべき問題がある．

まず，Web アプリケーション上で動作する JavaScript プログラムから直接永続時変値の値履歴を保持する時系列データベースへアクセスすることは困難である．そこで，本研究 JavaScript 言語を用いてバックエンドの処理を行える Node.js で中継サーバを立ち上げ，それを通して，時系列データベースへアクセスして永続時変値の値履歴を参照する．Web アプリケーションと中継サーバの通信は WebSocket で行う．

次に，そのような中継サーバを介した場合の，永続時変値の最新値をリアルタイムに取得する問題がある．WebSocket は非同期通信が基本となるが，一方で従来の SignalJ 言語は，時変値の値取得に，使用時に値を再計算するプル方式を採用しており，非同期通信とは相性が悪い．そのため，SignalJ の時変値の仕組みをそのまま本研究へ適用することができない．そこで，時系列データベース更新時に値を再計算し，それを JavaScript 上の永続時変値に通知するプッシュ方式を用いることで時変値の機能を実現した．

最後に，ブラウザ上でアプリケーションを動かすのであれば，言語を拡張するよりライブラリとして永続時変値を実装させることが望ましい．本研究では，SignalJ のような言語拡張を行わず，JavaScript のライブラリとして永続時変値の機構を実現する．

2. 永続時変値

本節では，永続時変値の現状について説明する．

2.1 時変値

時変値とはプログラムにおける変数の１つであるが，従来の命令型プログラムの変数とは異なっており，明示的な再代入を行わず時間とともにその値が変化する．例として，図 1 のプログラムを用いて説明する．これは SignalJ で書かれたプログラムである．SignalJ では時変値の宣言に `signal` という修飾子を用いる．

```
signal int a = 3;
signal int b = a * 2;
System.out.println(b);//出力結果：6
a = 5;
System.out.println(b);//出力結果：10
```

図 1 SignalJ における時変値

```
signal class Vehicle {
    persistent signal double x, y;//x座標，y座標
    signal double dx = x-x.last();//x座標の変化量
    signal double dy = y-y.last();//y座標の変化量
    signal double v = distance(dx,dy);//速度
    ...
    public Vehicle(String id, ...) { ... }
}
```

図 2 永続時変値の例

図では二つの時変値 `a` と `b` が宣言されている．`b` は `a` を 2 倍にした値になるように宣言されており，`a` は 3 で初期化されるので `b` の値は 6 で初期化される．`a` の値が 5 に更新されると，それに伴って `b` の値も再計算され，元の 6 のから 10 に更新される．

シグナルはもともと Fran[3] や FrTime[1] のような関数リアクティブプログラミング（FRP）言語で議論されてきた言語要素であるが，今では上述した SignalJ や REScala[9]，PuPPy[10] のように，命令的な言語要素を含む既存のプログラミング言語を拡張した実装も存在する．後者では，代入演算子などを用いて命令的にシグナルの値を更新することも可能である[*1]．

2.2 永続時変値

時変値は通常値が更新されるともとの値は消えてしまう．例えば上述した車輌追跡のできるようなアプリケーションを考えたときに，時変値の過去への遡及ができると便利である．この問題を解決したのが永続時変値である．永続時変値は時間経過による値変化を含めて抽象化した時変値のことである．時変値の機能に加えて，時変値の値が変化する度に時変値を時刻印と共に時系列データベースに保存することで，過去への遡及が可能となっている．SignalJ では，永続時変値は 2.1 節で述べた時変値の宣言に加えてさらに `persistent` の修飾子を追加することで宣言できる．

実際に永続時変値の機能を使った車両追跡プログラムを例にとって説明する（図 2）．

[*1] 更新できるのは通常，他のシグナルに依存しない，データフローの入り口に位置するシグナルのみである．

修飾子 persistent がついた時変値 x と y が永続時変値として宣言されており，クラス宣言時には修飾子 signal が必要となる．永続時変値 x，y が更新される度に dx，dy，v も再計算されるので永続時変値に依存している時変値も永続時変値として扱われる．

Vehicle クラス（signal という修飾子のついたクラス）には，以下のようなインスタンス生成時に必ず id が渡される（図 2 のコンストラクタで id パラメータは必須である）．

```
Vehicle v = new Vehicle("Oita-a1234");
```

その id と紐づいたテーブルが時系列データベース上に作られる（既に作られている場合は，そのテーブルと紐づけされる）．上記の Vehicle インスタンス生成時の例の場合は文字列”Oita-a1234”をテーブル名に含むテーブルが作られる．永続時変値の実体は言語処理系の外側にある時系列データベースにあり，このテーブルは Vehicle クラスの情報をもとに以下のスキーマを用いて生成される．

```
create table(time timestamp,
             x double,
             y double,
             dx double,
             dy double,
             v double);
```

Vehicle インスタンスの時変値においては，値が更新される度にそれぞれの時変値名と同じ名前の列に値を格納する．また，time 列には永続時変値の値が更新された時点での時刻が格納される．永続時変値 x，y の更新は set メソッドにより以下のように行われる．

```
v.set(xの座標値，yの座標値);
```

その際，dx，dy，v などの x と y に依存する値を全て更新したのち，上述したテーブルへの挿入操作が行われる．実際に参照したい過去の時刻の時変値を参照する場合には以下のように snapshot メソッドで時刻印を指定し，過去に遡る（これは set メソッドを呼んでいるのとは別の，読み込み専用のインスタンスで使うことを想定している）．

```
vr.snapshot("2021-12-02T11:22:33");
```

上の例では"2021-12-02T11:22:33"に Vehicle インスタンスの時刻が設定され，各フィールド x，y，dx，dy，v の値は設定された時刻印を用いて Vehicle インスタンス vr から参照される．

3. JavaScript における永続時変値の実装

本節では，JavaScript での永続時変値の実装方法について述べる．

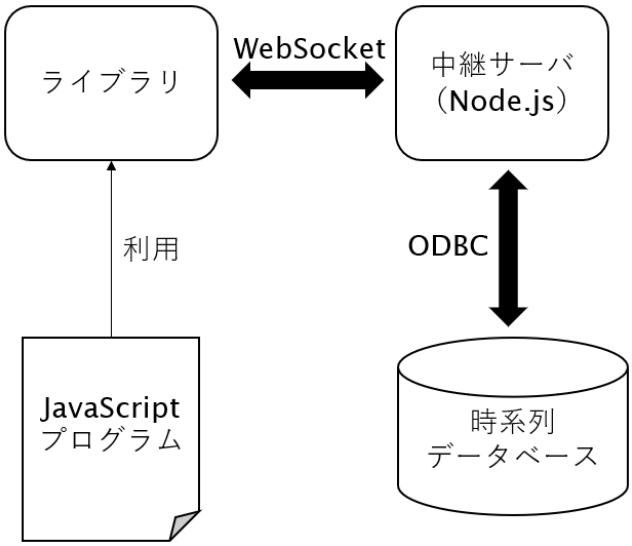

図 3 本提案のアーキテクチャ

3.1 実装の概要

本研究で JavaScript に永続時変値の機構を実装していくに当たり，時変値を保存する場所の時系列データベース，永続時変値の機構を実現する JavaScript のライブラリ，時系列データベースとライブラリを仲介する中継サーバの 3 種類が必要となる（図 3）．そのうち，時系列データベースは SignalJ のバックエンドとして用いられている TimescakeDB を使用する．よって，本研究を実現させる為に新たな開発が必要となるのはライブラリと中継サーバとなる．以下でそれらを説明する．

3.2 ライブラリのクラス構成

本研究では，ブラウザ上でアプリケーションを動かすことを前提としたので，永続時変値をライブラリとして実現した．永続時変値をライブラリとして実現することによって，ブラウザ上で動くアプリケーションからこのライブラリを呼び出すことで，そのアプリケーションに永続時変値の機能を提供することができる．

永続時変値をライブラリとして実現するにあたり，大まかに必要となる機能は４つである．それらは，まず，id を用いてライブラリが呼び出されたときに，その id に対応する時系列データベースに上の各構成要素を参照する各オブジェクトを提供する機能，次に時系列データベースに保存された時変値の履歴を参照する為にクエリを中継サーバに送信する機能，そして実際に参照したい過去の時刻を時刻印として指定する機能，最後に中継サーバ経由で得た時系列データベースに保存された時変値の値を取得する機能である．

その４つの機能を実装する２つのクラスを作成した（図 4）．１つは永続時変値の機能を実現した PSignal クラスと，もう 1 つは関連する永続時変値をひとまとまりにして扱う SignalClass クラスである．この２つのクラスは以下

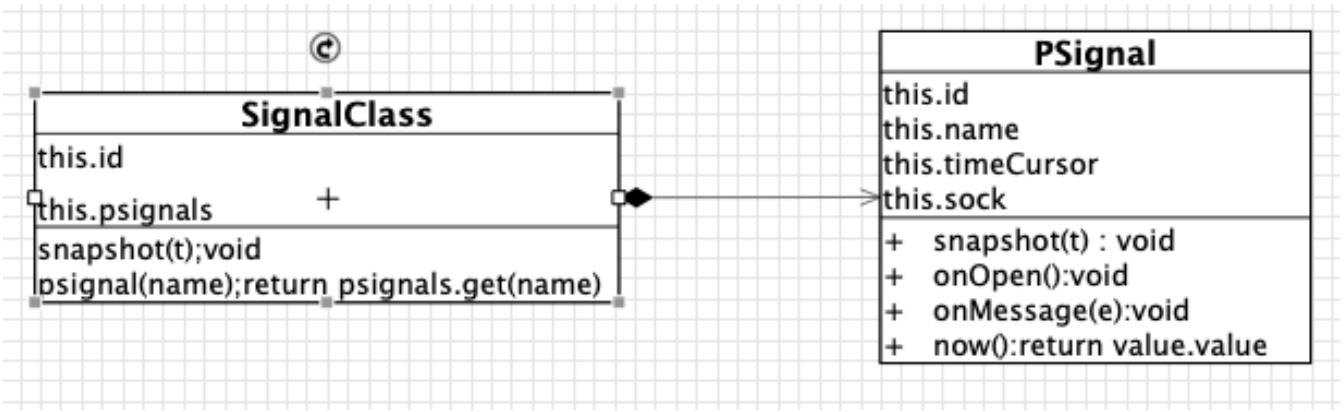

図 4 提案ライブラリのクラス構成

```
class SignalClass {
    constructor(id,...names) {
        this.id = id;//データベースのテーブル名
        this.psignals = new Map();
        for (const i in names) {
            this.psignals.set(
                names[i],
                new PSignal(this.id,names[i]));
        }
    }
    psignal(name) {
        return this.psignals.get(name);
    }
    snapshot(t) {
        // psignalsの全てに対し，時刻tを設定する
        this.psignals.forEach((item) => {
            item.snapshot(t);
        });
    }
}
```

図 5 SignalClass クラス

の各節で詳しく説明する．

3.2.1 SignalClass クラス

図 5 に，SignalClass クラスの詳細を示す．このクラスには，id を用いてライブラリが呼び出されたときにその id に対応する時系列データベース上の各構成要素を参照する各オブジェクトを提供する機能と，実際に参照したい過去の時刻を時刻印として指定する機能を実装した．提供されるメソッドは psignal と snapshot の 2 つとなる．SignalClass インスタンスを生成するには，引数として id(時系列データベースのテーブル名) とそのインスタンスに含めるべき永続時変値名のリストをコンストラクタに渡す必要がある．

コンストラクタは受け取った永続時変値名をキー，コンストラクタ内で生成される永続時変値名に対応する PSignal インスタンスを値として Map オブジェクトに格納する．Map オブジェクトを用いることによって，オブジェクト内の繰り返し処理が容易に行えるので，永続時変値の管理が容易となる．

psignal メソッドは呼び出されると，引数で渡された文字列をキーとして Map オブジェクトから PSignal インスタンスを取得し，それを返す．snapshot メソッドは引数として参照したい過去の時刻を時刻印として受け取り，その時刻印を Map オブジェクト内の PSignal インスタンス全てに対して適用する．

3.2.2 PSignal クラス

図 6 に，PSignal クラスの詳細を示す．このクラスでは，WebSocket を用いた中継サーバとの通信を用いて永続時変値の機能を実装した．PSignal インスタンスを生成するには，テーブル名と永続時変値に対応するテーブルの列名をコンストラクタに渡す必要がある．コンストラクタでは受け取った 2 つの引数と現在時刻を表す文字列を用いて，時系列データベースから時変値の更新履歴を参照する際に用いるクエリの初期化を行う．その後，中継サーバとの接続を WebSocket を用いて設定する．

snapshot メソッドは SignalClass インスタンスの snapshot から呼び出される．引数として参照したい過去の時刻を時刻印として受け取り，PSignal インスタンス内の時刻印を表すフィールドにその値を設定する．その後，onOpen メソッドを呼び出し，その時刻印で示された時刻における永続時変値の値を中継サーバにリクエストする．onOpen メソッドでは時系列データベースから時変値の更新履歴を参照する際に用いるクエリをオブジェクトにしてまとめる．その後，オブジェクトを JSON 形式に変換して，中継サーバへと送信する．

onMessage メソッドは中継サーバから非同期に応答を受け取った際に呼び出される．その際には，指定した時刻における時変値の値が JSON 形式として送られてくるので，JSON 形式からオブジェクトへと変換して，永続時変値の「現時刻」を示す this.value フィールドに受け取った値を代入する．

```
class PSignal {
    onOpen() {
        const json = {
            table_name:this.id,
            column_name:this.name,
            time:this.timeCursor
        };
        this.sock.send(JSON.stringify(json));
    }
    onMessage(e) {
        let rejason = JSON.parse(e.data);
        this.value = rejason;
    }
    constructor(id, name) {
        this.id = id;//idはテーブル名
        this.name = name;//nameはテーブルの列名
        this.timeCursor = 'now()';//時刻印
        …//WebSocketを用いて中継サーバと接続
    }
    snapshot(t) {
        this.timeCursor = t;//時刻印の設定
        this.onOpen();
    }
    now() {
        return this.value.value;
    }
}
```

図 6 PSignal クラス

```
const { Client } = require("pg");
const client = new Client(...);
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8000 });
client.connect();//データベース接続開始

wss.on('connection', function connection(ws) {
    ws.on('message', function incoming(message) {
        var queryobj= JSON.parse(message);
        querystring='SELECT time,
                ${queryobj.column_name} as value
            FROM ${queryobj.table_name}
            WHERE time <= '${queryobj.time}'
            ORDER BY time DESC';
        const query = {text: querystring,};
        client .query(query)
        .then((res) => {
            const reply = {
                table_name :'${queryobj.table_name}',
                value : res.rows[0].value,
                time: res.rows[0].time
            };
            var data = JSON.stringify(reply);
            ws.send(data);
        })
        .catch((e) => console.error(e.stack));
    });
});
```

図 7 中継サーバの振る舞い

now は外部のプログラムから呼び出される API メソッドである．このメソッドは this.value フィールドに設定された永続時変値の現在値を返り値とする．

3.3 中継サーバの実装

本研究では，ブラウザ上で実行されるライブラリと時系列データベースとのやり取りを，両者の間に中継サーバを挟むことによって可能にした．中継サーバの実装には Node.js を使用した．Node.js とはサーバサイドで JavaScript を実行できるようにしたものである．

本研究で開発した中継サーバは，時系列データベースとライブラリの両者と通信を行う．時系列データベースとの通信では，ライブラリから送られてきたメッセージからクエリを取り出してデータベース内に保存されている値の参照を行う．ライブラリとの通信では，ライブラリ側から送られてくるメッセージの受信と時系列データベースから参照した値の送信を行う．次に，中継サーバの挙動を説明する．

中継サーバの振る舞いを，JavaScript コードとして図 7 に示す．まず，中継サーバが立ち上げられると，予め用意しておくデータベースと接続する為の各設定情報を用いて時系列データベースへアクセスする．その後，ライブラリから時系列データベースへのクエリメッセージが JSON ファイル形式で送られてくると，そのメッセージを JSON ファイルからオブジェクトに変換する．変換後は，メッセージに含まれているテーブル名，永続時変値に対応するテーブルの列名，時刻印をもとに参照する値を時系列データベースから検索する．値が見つかればその値とそれが保存されていたテーブル名，その値が時系列データに保存された時刻の内容を含むオブジェクトを生成し，それを JSON 形式に変換してライブラリへ送信する．中継サーバは，サーバ自体が閉じられるまで，ライブラリからメッセージが送られる度にこの一連の処理を行う．

3.4 永続時変値の更新通知

永続時変値が最初に提案された SignalJ は，プル型の評価を前提としており，値が必要になった際に評価が行われる．そのためには，値をリクエストしてからデータベースが値を返すまで実行をブロックする同期の機構が必要になるが，本研究のように間に WebSocket による通信を挟むとその実現が難しくなる．そこで本研究では，（他の永続時変値の更新を通じて）時系列データベースが更新されると，それを各 PSignal インスタンスに通知するプッシュ型

の値評価方法を取り入れた．

プッシュ方式の永続時変値の実現方法では，まず時系列データベース側に，データ更新を通知する機能があることが前提である．バックエンドに使用している TimescaleDB は PostgreSQL の拡張なので，PostgreSQL に提供される Notify 機能を使ってそれを実現できる．具体的には，テーブルに対して INSERT 文が実行されたときに Notify イベントを発火するトリガを CREATE TRIGGER 文を用いて作成すればよい．中継サーバにおいては，Node.js から予め LISTEN 文を実行し，Notify イベントを購読しておく．Notify イベントが通知されたら，再度上述した方法で時系列データベースに問合せを行い，結果をライブラリに通知することで，プッシュ方式の永続時変値を実現した．

4. 検証

本節では，提案ライブラリと中継サーバの振る舞いを確認するために実際に行った作業を述べる．

まず，ブラウザ上で動くサンプルプログラムを作成する．文献 [11] で用いられた永続時変値をテスト用データとして用い，サンプルプログラムからその永続時変値にアクセスする．サンプルプログラムでは，snapshot を用いて永続時変値の時刻印を変更する操作を実装する．テスト用データとして用いる時系列データの内容は予めわかっているので，期待値と結果が等しければ動作確認がとれる．

4.1 サンプルプログラム

本起動実験で使用するサンプルプログラムとして，ブラウザ上で動く Web アプリケーションを作成した（図 8）．これはテキスト入力欄とボタンを使用することによって，永続時変値に関する情報の設定と，それの値の参照を行えるものである．

このサンプルプログラムは, 図 9 に示す 3 段構成の HTML 文書を参照する．まず，1 段目は input タグが二つと参照というラベルのついたボタンの構成となっている．id という id のついた input タグには参照したい永続時変値が属する id を入力し，name という id のついた input タグには永続時変値の名前を入力する．参照ボタンを押下すると，SignalClass インスタンスがそれぞれの入力文字列を第一，第二引数として生成される．その後，第二引数を用いて PSignal インスタンスが生成される．

2 段目は 1 つの input タグと時刻設定というボタンで構成されている．ここの input タグでは永続時変値の時刻印を設定する．ボタン押下によって，1 段目で生成した SignalClass インスタンスの snapshot メソッドが呼ばれ，指定した時刻印が設定される．

3 段目は値取得というボタン 1 つの構成となっている．これを押下すると now メソッドにより永続時変値の値を取得し，それがブラウザ上のコンソールに出力される．

```
var signal = null;
var re = null;

document.addEventListener('DOMContentLoaded',function(e){
  document.getElementById('refbutton').
    addEventListener('click',function(e){
      let id = document.getElementById("id");
      let name = document.getElementById("name");
      signal = new SignalClass(id.value, name.value);
      re = signal.psignal(name.value);
    });
});
document.addEventListener('DOMContentLoaded',function(e){
  document.getElementById('timebutton').
    addEventListener('click',function(e){
      signal.snapshot(
        document.getElementById("time").value);
    });
});
document.addEventListener('DOMContentLoaded',function(e){
  document.getElementById('submit').
    addEventListener('click',function(e){
      console.log(re.now());
    });
});
```

図 8 サンプルプログラム

```
<!-- 1段目：永続シグナル参照 -->
<input type="text" id="id">
<input type="text" id="name">
<input type="button" id="refbutton" value="参照">
<p></p>
<!-- 2段目：時刻印設定 -->
<input type="text" id="time">
<input type="button" id="timebutton" value="時刻設定">
<p></p>
<!-- 3段目：値取得 -->
<input type="button" id="submit" value="値取得">
```

図 9 サンプルプログラムの HTML

このサンプルプログラムを実行し，時刻印の変更や永続時変値の値更新をそれぞれ試した後，値取得ボタンの押下によってコンソール上に期待通りの値が出力されることを確認した．

5. 関連研究

本節では，本研究の関連研究として，JavaScript に基づく FRP 言語や，FRP の Web アプリケーションへの応用に関する研究について紹介する．

Flapjax[7] は JavaScript に基づく FRP 言語であり，本研究と同様の議論が可能であると考えられる．Flapjax は

時間変化する値を，Event（離散時間上の値のストリーム）と Behavior（連続時間上を自動的に変化する値）の二種類の言語要素を用いて抽象化する．同様の言語要素を Web アプリケーションに応用する取り組みとして文献 [8] もあげられる．それに対して本研究では，時間変化する値としてシグナルのみを用い，データフローの入り口（命令的な更新が許可される）とそれ以外を区別するアプローチをとっている．同様のアプローチは SignalJ[4] や PuPPY[10] にも見られる．このアプローチは実装が容易であり，本研究のようなライブラリを実装するのに適している．また，RxJS[*2]はシグナルや FRP と似た機能を提供するが，RxJS ではグリッチ（値更新時に起こる一時的な不整合）が発生することが知られており，厳密にはシグナルとは異なるものである．

Denuzière らは，FRP をサーバサイドも含めた Web アプリケーションフレームワークに適用する仕組みを提案している [2]．それに対し，本研究は主にフロントエンドアプリケーションを対象としており，バックエンドのサービスとして時系列データベースがあることが前提である．

これらの研究のいずれにおいても，時間変化する値（シグナル，またはそれに類するもの）を永続化し，それを Web フロントエンドアプリケーションの一部として利用することは実現されていない．

6. まとめ

本研究では，現在，SignalJ で実現している永続時変値の機構を言語拡張せずに，ライブラリと中継サーバを開発することによって，JavaScript で実現する方法を提案した．動作確認の結果，開発した JavaScript のライブラリと中継サーバを用いることによって，永続時変値の時刻印を設定することで，過去に時系列データベースに保存された値を参照するという機構を実現することができた．なお，本研究では永続時変値を実現するライブラリの提案を行ったが，永続でない時変値については議論していない．永続でない時変値を JavaScript 上で実現する取り組みについては，上述した関連研究のほかに，西津らの取組みも知られており [12]，これらの成果を本研究に取り入れていくことは今後の課題である．

謝辞 本研究の提案内容について深くご議論いただいた東京工業大学の増原英彦氏と株式会社豆蔵の青谷知幸氏に感謝する．本研究は科研費研究課題 21H03418 の支援を受けたものである．

参考文献

[1] Cooper, G. H.: Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language, PhD Thesis, Department of Computer Science, Brown University (2008).
[2] Denuzière, L. and Granicz, A.: Enabling modular persistence for reactive data models in F# client-server Web applications, MODULARITY Companion'16, pp. 55–64 (2016).
[3] Elliott, C. and Hudak, P.: Functional Reactive Animation, Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97), pp. 263–273 (online), DOI: 10.1145/258949.258973 (1997).
[4] Kamina, T. and Aotani, T.: Harmonizing Signals and Events with a Lightweight Extension to Java, The Art, Science, and Engineering of Programming, Vol. 2, No. 3 (online), DOI: 10.22152/programming-journal.org/2018/2/5 (2018).
[5] Kamina, T. and Aotani, T.: An Approach for Persistent Time-Varying Values, Onward!'19, pp. 17–31 (2019).
[6] Kamina, T., Aotani, T. and Masuhara, H.: Signal Classes: A Mechanism for Building Synchronous and Persistent Signal Networks, ECOOP 2021, LIPIcs, Vol. 194, pp. 19:1–19:30 (2021).
[7] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A. and Krishnamurthi, S.: Flapjax: A programming language for Ajax applications, Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09), pp. 1–20 (online), DOI: 10.1145/1640089.1640091 (2009).
[8] Reynders, R., Devriese, D. and Piessens, F.: Experience report: functional reactive programming and the DOM, Proceedings of Programming'17, pp. 1–6 (2017).
[9] Salvaneschi, G., Hintz, G. and Mezini, M.: REScala: Bridging between object-oriented and functional style in reactive applications, Proceedings of the 13th International Conference on Modularity (MODULARITY'14), pp. 25–36 (online), DOI: 10.1145/2577080.2577083 (2014).
[10] Zhuang, Y.: A lightweight push-pull mechanism for implicitly using signals in imperative programming, Journal of Computer Languages, Vol. 54 (2019).
[11] 上野 颯太，紙名 哲生: 永続時変値の分散化に向けた基盤システムの試作, 第 210 回ソフトウェア工学研究発表会 (2022).
[12] 西津 佑真，紙名 哲生: シグナルに基づくマイクロフロントエンドアプリケーションの実現法, 第 137 回情報処理学会プログラミング研究会 2021-4-(3) (2022).

[*2] https://rxjs.dev/